# Modern Atari System Software

This reference guide, available separately, is an invaluable source of documentation for any serious programmer on the Atari 680x0 platform. It contains a wealth of information on using the newer Atari TOS-based operating systems from within C, BASIC and assembly language in addition to many general programming hints and tips.

Topics covered include:

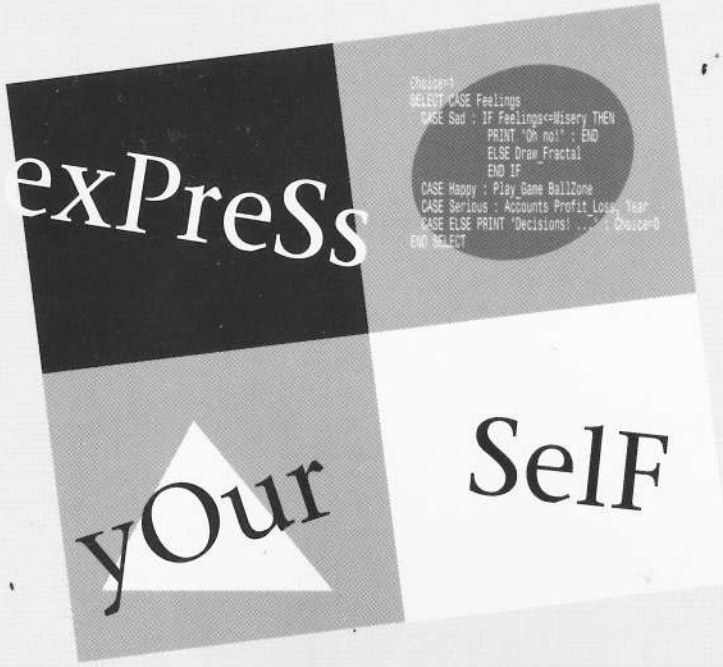
- An overview of the Atari Falcon030 computer including its video, audio and DSP sub-systems and the new system calls needed to handle this exciting hardware
- Details of how to program with MiNT™, MultiTOS™ and SpeedoGDOS™
- Information on using the new features of TOS 4.0, including all the AES enhancements, the Cookie jar, 3D dialogs etc.
- The Atari Style Guide is included which gives advice on programming a consistent user interface so as to improve the look and ease-of-use of your programs
- Technical Appendices cover the SpeedoGDOS™ header, MultiTOS™ configuration, operating system bindings and the various error codes/signals that can be detected

*Modern Atari System Software* details the interface code for calling all these new functions from Lattice C 5.60, HiSoft BASIC 2.10 and HiSoft Devpac 3.10 and is an essential buy for all Atari programmers. Price £24.95; available from all good bookshops or, in case of difficulty, directly from HiSoft.

*HiSoft BASIC 2.10 for ST/STE/TT/Falcon Computers*

**HiSoft**
**High Quality Software**

# HiSoft BASIC
## version 2.10




# Addendum

*more software for your Atari ST/STE/TT/Falcon from*

**HiSoft**
**High Quality Software**

# Table of Contents

# *Introduction*

This manual outlines the improvements in HiSoft BASIC 2.10 for the Atari compared with the original version HiSoft BASIC 2.0.

The new library calls for Falcon TOS, SpeedoGDOS, MiNT and MultiTOS are detailed in the book, *Modern Atari System Software*. They are accessed via the FALCON, SPEEDO, MINT and GEMAES libraries respectively.

In summary the other improvement are as follows:

- Extended dialog routines with keyboard shortcuts and pop-up menus.
- Named compiler options.
- Improved code generation.
- Improved debugger.
- MultiTOS friendly dynamic heap option.
- Inline GEMDOS/XBIOS/BIOS code generation.
- Support for the Falcon/MultiTOS 3D effects in dialogs.
- Long integer constants are now supported. e.g. `CONST fred&=123456789`.
- Buildlib now includes the Speedo, MiNT and Falcon libraries when `-q` is used.
- The `PALETTE` command now works on the Falcon in modes with up to 16 colours. In 256 colour mode it is best to use the VDI `vs_color` command - the `PALETTE` statement will give you an `Illegal function call` error.
- Bindings for the `appl_trecond` and `appl_tplay` in the `GEMAES` library.
- Bindings for the `vq_curaddress` in the `GEMAES` library.

# The GEM Toolkit Extensions

## Extended Form Handler

`Xform_do` is a collection of high level routines written in BASIC which you can `REM$include` (or `'$include`) into your programs to produce more friendly and usable dialog boxes. The routines are compatible with the built-in form handler but have a few additional features. The `xform_do` routines require the file TOOLBOX.BAS.

- Every exit object can have a keyboard shortcut equivalent. This is indicated in the dialog by an underlined character. Keyboard shortcuts use the Alternate+key combination and are defined using WERCS. Keys A to Z are supported.
- In addition to the Return key returning the default object, the Undo key will return your chosen object. This would normally be a Cancel button.
- The left and right cursor keys can be used in combination with the shift keys to jump to the beginning or end of a text field in `G_FTEXT` and `G_FBOXTEXT` objects. Tab moves the cursor to the next editable object as normal, and Shift+Tab also moves backwards.
- If required, you can enable the right mouse button when in the form handler. If the global variable `mouse_detect_both` is set to non-zero the right button will be enabled. The AES does not handle the right button exactly the same as the left. The best use for the right button is probably to increment or decrement numerical text objects as shown in the example program XFRMTEST.BAS.



## The Routines

### FUNCTION xHandleDialog(BYVAL editnum, BYVAL cancel_obj, button)

This general routine can be used in place of `HandleDialog`. The return result is the exit object number. The parameter `editnum` gives the text field where the cursor will be positioned initially; if you don't have any editable text fields then pass the value 0. The `cancel_obj` parameter is the object number that will be returned when the Undo key is pressed. The additional button parameter will contain the mouse button mask in the usual way. This is only useful if you need to know which button was pressed, otherwise just pass 0. To enable right button detection set the global variable `mouse_detect_both` to non-zero.

`xHandleDialog` calls `xform_do` and `xobjc_draw` which are documented below. Note that the `tree&` parameter is not needed when calling these routines.

### FUNCTION xform_do(BYVAL editnum, BYVAL cancel_obj, button)

The actual form handler. This routine behaves very much like the OS form handler but allows for additional features. Unlike `form_do`, there is no need to pass the tree address, as this is an HGT global. The `cancel_obj` parameter should be the object number you want returned when the Undo key is pressed. The parameter `button` will contain the button bit mask if the global `mouse_detect_both` is non-zero.

### SUB xobjc_draw(BYVAL object, BYVAL depth, BYVAL x, BYVAL y, BYVAL w, BYVAL h)

This sub-program replaces the AES `objc_draw`. It is fully compatible with the former but draws underlines for the 'hot-keys'. Using a depth of 10 will cause this routine to show all objects, thus speeding up the display a little.

### *SUB xobjc_change(BYVAL object, BYVAL x, BYVAL y, BYVAL w, BYVAL h, BYVAL newstate, BYVAL flag)*

This subroutine behaves in the same way as `objc_change`, but it also redraws the underline if a hotkey exists and `flag` is non-zero.

## *Assigning keyboard shortcuts to objects*

An object must be selectable to have a 'hot key'. It does not have to be an exit object, but it must have the selectable bit set. The simplest way to define a hot key is with WERCS. Select the object and press Alternate+E or select the Extras item from the Flags menu. The ASCII value of the hot key goes in the extended type field (so use 65 for A, 66 for B etc.). At present, WERCS does not support the new form handler so you will not be able to test the hot keys from within WERCS itself.

## *Form windows*

Form windows provide an easy way to produce dialogs in windows that can be moved about by the user whilst still using the facilities of conventional dialog boxes. This is a recommended software design approach under MultiTOS since it enables the user to switch applications whilst a dialog is displayed unlike old fashioned modal dialog boxes.

Form windows also give the user the ability to display several dialog boxes at once. The disadvantage from the programmer's point of view is that you need to structure your code so that you can cope with an exit from a dialog at any time, but this is a price worth paying for the improved usability it provides.

In addition you can use the keyboard shortcuts of the `xform_do` routines in form windows. These keyboard shortcuts take precedence over any menu keyboard shortcuts when the window is on top.

The form library requires the TOOLBOX.BAS, WINDOW.BAS, OBJWIND.BAS and XFORMDO.BAS files of the BASIC toolbox.

When designing dialog boxes to go in form windows it is best to use a simple 1 pixel wide border. Outlined boxes look odd inside windows.

The XADDRESS.BAS example file shows form windows in action.



### *FUNCTION OpenFormWindow%(windowname$, BYVAL WindowComp, BYVAL treeno, BYVAL startobj, BYVAL cancelobj, BYVAL closeproc&)*

This is the subroutine that opens a form window. The `windowname$` parameter gives the title of the window. The `WindowComp` parameter gives which components of the window furniture to display. The recommended value for the latter is `win_name+win_close+win_move` although you may wish to use other values for special purposes. Note that the routines do not support scrollable form windows.

`treeno` gives the resource file index of the object tree to be displayed in the window. `startobj` gives the text item to be edited when the form window is opened. Use 0 if there are no editable text fields in this dialog box.

`cancelobj` is the object that will be returned if the window close box is selected or the `Undo` key is pressed when the dialog is the top window.

`closeproc&` is the address of the sub program that will be called when the window is closed or an exit button or its keyboard shortcut is pressed. If this routine does nothing, the form window will simply be closed.

This routine can use the following shared variables:

`commonobj`
: this gives the index of the object that was selected.

`commonclose`
: if this variable is set to 0 then the form window will not be closed, thus keeping the dialog open for new key-presses and button clicks. In this case you will need to re-draw the parts of the dialog box that you have updated including the exit object The `FullRedraw` and `Redraw` sub-programs of WINDOW.BAS can be used to do this. See the examples on disk for further information.

### *SUB object_redraw(BYVAL obj)*

When using form windows you may find that you need to redraw a single object in a window because it has changed. This routine will do just that. It handles any border that may be outside the co-ordinates of the object and also ensures that the display is not corrupted if only part of the object is visible. This routine *requires* the OBJCEXT.BAS file described below.

See the XADDRESS.BAS example for a demonstration of how to use this sub-program.

## *Pop-up menus*

The new Falcon and MultiTOS AES provide pop-ups built-in but what do you do if you want your program to run on older machines?

### *FUNCTION popup(BYVAL wind_handle, BYVAL treenum, BYVAL prev_pop, BYVAL x,BYVAL y)*

`Popup` is a function which takes five parameters. The first is the current AES window handle which is what `OpenFormWindow` returns. If you are using standard, modal, dialog boxes simply pass 0 as the window handle. The second parameter is the object tree number as per the WERCS.BH file. The `prev_pop` parameter should normally be the value previously returned from this function. It is used to place a check on the previously selected item. If you do not want a check, just pass 0. The last two parameters are the x and y co-ordinates for the top left corner of the popup menu.

The function ensures that the menu is not displayed outside the current window, or outside the screen display if no window handle is passed. The function returns the object number that was selected by the user. It requires the OBJCEXT.BAS routines.



## *Creating an object tree for a popup menu*

A popup menu object tree must consist of `G_STRING` objects bound in a box with a border size of -1. We also recommended the box is shadowed. You may not use any other object types.

See the example program, POPDEMO.BAS, for more information.

## *Additional object routine*

### *SUB objc_extent(BYVAL object, VARPTR x, VARPTR y, VARPTR w, VARPTR h)*

This will return an AES rectangle corresponding to the visual representation of the object. It takes the new 3D object effects into account if running on a version of the operating system that supports them. It is used by the form window and pop-up routines.

# *Changes to WINDOW.BAS*

The window routines have been improved so that you can have more than 7 windows if the operating system supports this and so that fewer redraws are generated. There have also been internal changes to support form windows.

Windows are now byte-aligned by default. This gives faster display of text in text and form windows. You can supress this, so that windows may be moved to any horizontal pixel position by setting the variable `supress_byte_align` to -1.

### *SUB TopAWindow(BYVAL WindowHandle)*

This routine should be called if you wish to bring one of your windows to the front. It will redraw the cursor if the window is a form window in addition to the update caused by `wind_set(WF_TOP)`. There is an example of its use in XADDRESS.BAS.

# New routine in TOOLBOX.BAS

### aes_version

This function returns the version of the AES that is currently running. It can be used to give extra facilities where the operating system supports them whilst still running when they are not. For example the `FileSelect$` routine uses this to find whether the prompt string is supported.

# Other toolbox improvements

The `FileSelect$` routine now uses an additional shared variable `fsmessage$` which will be displayed as the file selector's title if the operating system supports this.



# Named Compiler Options

HiSoft BASIC 2 now supports names for the compiler options as well as the rather cryptic letters and symbols used by previous versions. The names may be used in `REM $` statements and following a + character on the command line. Options may be switched off by using `NO` at the start of the name. e.g.

```
REM $ARRAY
```

will switch array bounds checking on, whereas

```
REM $NOARRAY
```

will switch array bounds checking off. The line

```
REM $loadbits 7
```

in a program will set the program's prgflags so that it can be loaded into TT RAM, may use TT RAM and won't bother clearing the rest of RAM when it starts up. Note that options that have parameters (like `$loadbits` above) use a space as the separator between the name and the parameter.

Similarly using a command line of

```
test +hclndebug +debug +extdebug
```

with the standalone version of the compiler, will compile the program test with full debugging information for Mon.

```
REM $JUMPS
```

will re-instate jumps between subroutines if these have been switched off.

The following table gives the complete set of options, with the new name, the old form, a description and the dialog in which they appear. The new names shown in this table are the names to use to override the standalone compiler's defaults.

| New | Old | Description | Dialog |
|---|---|---|---|
| LOADBITS | ^ | Output prgflags | output |
| DISABLE | ~ | Remove reserved words | other |
| EXPORTS | ! | Enable variable exports | standard |
| NOFNSINLIBS | # | No "FN"s in libraries | standard |
| TOKENISE | $ | Dump Tokens | other |
| NOJUMPS | % | Remove inter-sub-program jumps | advanced |
| HCLNDEBUG | & | Add Mon line numbers | debug |
| OLDDOUBLES | * | Use old style doubles in files | advanced |
| QUIET | . | Suppress compiler titles | other |
| TOKENS | @ | Pre-tokenised file | standard |
| NOAUTODIM | [ | Array Warnings | standard |
| UNDEFSUBS | ] | Allow undefined sub-programs | advanced |
| ARRAY | A | Array Checks | standard |
| BREAK | B | Break Checks | standard |
| BATCH | C | Continue on compiler errors | advanced |
| EXTDEBUG | D | Use Extended Debug | debug |
| ERRORS | E | Error Messages | standard |
| TO | F | Output Filename | standard |
| GEM | G | Force GEM program | advanced |
| LABELS | H | Maximum Labels | advanced |
| DESKACC | J | Desk Accessory Size | advanced |
| KEEP | K | Keep Size | advanced |
| LEAVE | L | Leave Size | advanced |
| MATHSTACK | M | Maths Stack Size | advanced |
| LINES | N | Add Line Numbers | standard |
| OVERFLOW | O | Overflow Checks | standard |
| PAUSE | P | Pause Checks | standard |
| HEAPDYNAMIC | Q | Dynamic Heap Size | advanced |
| RETSTACK | R | Return Stack Size | advanced |
| DEBUG | S | Add Debug information | debug |
| STRINGS | T | Temporary String Descriptors | advanced |
| UNDERLINES | U | Underlines in variables | standard |
| VARCHECKS | V | Variable Checks | standard |
| NOWARN | W | Suppress Compiler Warnings | advanced |
| STACK | X | Stack Checks | standard |
| NOWINDOW | Y | Suppress Default Window | advanced |
| NOWAIT | Z | Finish without waiting for a keypress | other |



# *New Dynamic Heap*

Traditionally HiSoft BASIC has used a single area of memory for the storage of all the arrays and strings in your program and the size of this is determined when your program starts running (either a fixed amount using the Keep or the largest available block minus a set amount with the Leave option). This scheme has the advantage that the program's memory can not become fragmented - the BASIC runtimes can move arrays and strings in memory ensuring that the largest possible part of the heap is available for a string or an array.

Under MultiTOS this is not a very good way to behave. If your program takes all of the available memory bar a few Kb then the user certainly won't be able to start any other programs. Conversely if you make your program only keep a small amount of memory, your program may run out of memory especially when processing a lot of data. Also programs with a lot of string handling will run faster if they have more memory as they won't need to garbage collect as frequently (or perhaps not at all).

Traditionally HiSoft BASIC programs have only been able to use TT RAM or ST RAM but not both at once for their variable storage.

Hence a third, dynamic, heap option was born. Rather than allocating all its memory from the system at once it only allocates a little at a time. Small strings get their own heap as before; small arrays are stored in another area together and large arrays and large strings are allocated individually. However if the area for small strings or small arrays fills up, the BASIC runtimes can now just allocate some more memory from the system. Conversely if a program needs less memory than before, the extra memory is released to the system letting other programs have that memory if they wish.

In order to use the minimum amount of memory possible the BASIC runtimes should garbage collect every time one of these 'mini-heaps' fills up. At the other extreme, garbage collection could be delayed until the last possible moment when all the available memory has been used, but this would mean that the available free memory would fluctuate wildly and at times you wouldn't be able to start other programs.

To provide a middle path between these extremes the new option has an associated memory size. If the program hasn't yet used this total amount of memory when a mini-heap fills up then a new mini-heap is allocated without performing a garbage collect. If the program has already used this amount of memory then it will do a garbage collection first - only allocating a new heap if absolutely necessary.

This maximum memory value can be manipulated when the program is running by modifying the long value at `SYSTAB+76`. The new option is option `Q` (or `$HEAPDYNAMIC`) and may be set using the Advanced sub-box from the Options menu.

There are two disadvantages of the new scheme. The first is that it is possible for memory to become fragmented i.e. it is possible that there is 400K of free memory in the system but no one block is bigger than 50K and so DIMensioning an array of more than 50K will fail. This can't happen with the old scheme. It is only likely to effect programs that are left running for a long time and are allocating and de-allocating arrays or large strings - a Bulletin Board System for example.

Also some programs that will just run in a given memory size under the old scheme won't quite run with the new one, because there is normally some unused space at the end of the string and array mini-heaps. Games that take over the whole machine are an example of this.

If all this sounds complicated, that's because it is! However, for most purposes, using the new dynamic heap with a size of 300 will mean that your program will behave nicely under MultiTOS but will take as much memory as it needs if the user of the program asks for it.

Note that you can still use the Keep and Leave options if you wish.



# *Inline Trap Calls*

The compiler can now generate inline operating system calls to traps used by GEMDOS, the BIOS and the XBIOS without using library code. As a result you only get the code in your program if you use a routine. Normally you won't need to use this facility directly as the libraries provided will do this automatically but you may wish to if you want to produce the smallest code possible or for a new operating system extension. The syntax is as follows:

***DECLARE [SUB | FUNCTION ]***
***routine_name[(parameter_list)] LIBRARY***
***trap_number, call_number***

The function or parameter name and parameters take exactly the same form as a BASIC routine. The compiler generates code to push any parameters on the stack in reverse order (using the size as declared) and then pushes the `call_number` on the stack and performs a `TRAP #trap_number` instruction and adjusts the stack. Don't use strings or arrays as parameters as these will be passed as descriptors which the operating system won't understand. For example,

```
DECLARE SUB dfree(BYVAL buffer&, BYVAL driveno%)
LIBRARY 1,&h36
```

declares the sub-program `dfree` to call GEMDOS (trap 1) with function number 36 hexadecimal. The routine has two parameters: `driveno` a 16 bit value and `buffer` which is a 32 bit value giving the address of the buffer to be filled.

The assembler syntax to incorporate these in libraries is:

```
inl_sub name,trapno,callno,paras...
```

for subroutines,

```
inl_int name,trapno,callno,paras...
```

for functions returning a 16 bit integer, and

```
inl_lng name,trapno,callno,paras...
```

for functions returning a long integer.

Routines that require strings, or arrays as parameters still need explicit code and this will be included whenever the library is used, regardless of whether the individual routine is used.

# Mon 3 Improvements

This section outlines the differences between the version of the debugger supplied with the original HiSoft BASIC and that shipped with version 2.10.

The debugger is *always* called MON.PRG (rather than ...ST, ...TT etc.), although on the master disks the '030 version is known as MON030.PRG which the installation program normally re-names upon installation.

The major improvements are the screen mode handling (for Falcon) and the debugger window handling including improved source support.

## Mon's Windows

Up to five windows can be shown simultaneously or, by changing the width and height of the windows, you can show just two.

Each window is numbered from 1 to 5 and can display different types of information - window 1 can be of any type, register, memory, source code or disassembly; windows 2 and 4 can be memory, disassembly or source code windows whilst windows 3 and 5 are restricted to being memory windows.

## Stacking Windows

Each window also has depth - you can stack views beneath a window so that you have almost limitless flexibility in what you choose to display.

In addition you can *split* and *widen* most windows; split means to grow or to shrink the window vertically whilst widen means to do the same horizontally. These operations may hide other windows temporarily or they may uncover hidden windows.



## Locking Windows

Each window may now be locked to an arbitrary expression. Thus, you can lock a memory window to a register so that it displays the contents of the memory addressed by that register. Or you might want to lock a disassembly window to the PC, which is the default condition for window 2.

Each view on the window stack can be locked to a different expression although it does not make sense to lock the register window.

All the above window features will be discussed in more detail later.

# Mon Reference

## Numeric Expressions

The following is a an up-to-date list of the debugger's operators:

| Precedence | Operator |
|---|---|
| 1 | unary minus (-) and plus (+), source operators (# and ?) |
| 2 | bitwise not (~) |
| 3 | shift left (<<) and shift right (>>) |
| 4 | bitwise And (&), Or (!) and Xor (^) |
| 5 | multiply (*) and divide (/) |
| 6 | addition (+) and subtraction (-) |
| 7 | equality (=), less than (<), greater than (>), inequality (<> and !=), less than or equals (<=), greater than or equals (>=) |

### Source Operators

There are two operators which allow debugging at a source code level; these are the # and ? operators.

To use these operators, you must have a source window open which is associated with the loaded executable program. In turn, this loaded program must have been produced by a package that attaches line number information to the program. Otherwise the # and ? operators are invalid.

The # operator takes a source line number as its argument and returns the associated memory address, within the loaded program. So, say you have the source of hello.bas loaded into window 2 and the executable of hello loaded as the current program then:

```
m3=#20
```

will set the start address of window 3 to the address of line number 20 of the hello program (assuming that window 3 is not locked to another expression).

If the line number is out of range of the source (e.g. if you ask for line number 100 when there are only 90 lines of source), the result will be the address of the first or last line of the source, accordingly. If you use the # operator when there is no line number information available, the result will be 0.

The ? operator is the reverse of #; it returns the source line number, given a memory address. If the address is out of range of the code connected with the source window, ? returns a value of 0.

If you have only one source file loaded, the use of these operators is unambiguous. However, if you have loaded two or more source files into Mon's windows, # and ? may return unpredictable results; in this case it is best to use them when one source file is open in the current window - they will then relate to this file.

These operators allow you to perform a variety of commands on a source level such as: Set Breakpoint, Run Until and Lock Window. This can make the process of debugging a complex program a far simpler and less tiresome task.



## *Window Types*

There are five possible windows within the Mon display and the exact contents of these windows and how they are displayed is detailed below. The allowed types of each window are:

| Window | Allowed Types |
|---|---|
| 1 | register, memory, disassembly, source |
| 2 | memory, disassembly, source |
| 3 | memory only |
| 4 | memory, disassembly, source |
| 5 | memory only |

A window can have a number of different views attached to it; you can think of the window as a *stack*, having depth.

So, in window 2, you can view a disassembly of code, a section of memory and a portion of an ASCII file, although only one of these at a time is visible. To cycle through the different views use the Next/Previous View commands and to create or delete a display use the Open View and Close View commands.

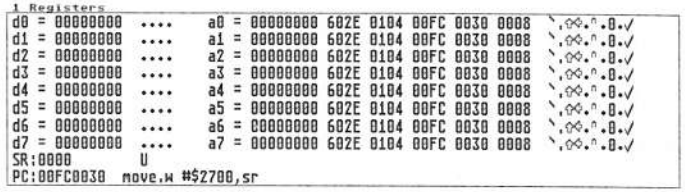
Most windows can also be *split*, either vertically or horizontally so that more, or less, can be displayed within the window - this action may hide or reveal other windows and it is best to experiment with the split commands (described below) to understand how they work.

A window can be locked to an expression so that its start address is dependent on the value of that expression - see the Lock to Expression command below.

You can also *zoom* a window; it will then occupy the whole of Mon's screen.

Each type of window will now be described.

## *Register Window*




the register window

The data registers are shown in hex, together with the ASCII display of their four bytes. The address registers are also shown in hex, together with a hex display of the memory that each register is addressing. This is word-aligned or byte-aligned as necessary, with non-readable memory displayed as **. To the right of this hex display is its ASCII interpretation.

The status register is shown in hex and in flag form, additionally with U or S denoting user- or supervisor-modes.

The PC value is shown together with a disassembly of the current instruction. Where this involves one or more effective addresses these are shown in hex, together with a suitably-sized display of the memory they point to.

For example, the display

```
tst.l $12A(a3) ;00001FAE 0F01
```

signifies that the value of $12A plus register A3 is $1FAE, and that the word memory pointed to by this is $0F01. A more complex example is the display

```
move.w $12A(a3),-(sp) ;0C001FAE 0F01 >0002AC08 FFFF
```

The source addressing mode is as before but the destination address is $2AC08, presently containing $FFFF. Note that this display is always of a suitable size (MOVEM data being displayed as a quad-word) and when pre-decrement addressing is used this is included in the address calculations.

The floating point registers (if present) are then displayed followed by the supervisor and Mon's memory registers.

The number of lines displayed in the register window may be altered using Control-← and Control-→; in addition Alt-F (to change the font) will allow you to view the floating point registers.



## *Source Window*

A new and very useful feature of the source window is that as low level code is single-stepped or displayed, the source display will also be updated to keep the relevant high level text on display. The top line of the source text is that which generated the code currently being stepped or pointed to by the program counter.

# *Window Commands*

Commands that are reached through the use of the Alt- key are normally available at any time. Many of these commands are connected with and apply to the *current window*. The current window is denoted by having an inverse title and it can be changed by pressing Tab or Alt- plus the window number.

Most window commands work in any window, zoomed or not, though when it does not make sense to do something the command is ignored.

The exceptions to the above are the Stack, Unstack and View Stack commands which, for ease of use, are not reached through the Alt- key and do not work on a zoomed window.

### Alt-L *Lock to Expression*

This allows source (with line number information), disassembly and memory windows to be locked to a particular expression. After any exception the start address of the display is re-calculated, depending on the locked expression. Each stacked view within a window can have its own lock.

Mon will ignore you if you try to lock a source window that refers to a program that does not have line number information attached to it.

If you try to lock a source window to an expression that lies outside the address range of the source file you will be ignored. This, in fact, is very useful; it means that if you have a stack of source windows (see below for details of stacking windows) which make up the executable that you are debugging and you lock each display to the PC, you will be able to trace the path of the program through each source file.

If an instruction in the top view calls a subroutine in the stack, the top view will not change but, if you then view the relevant stacked view, it will change to show you the called subroutine.

To unlock, simply enter a blank string.

You can lock one window to another window by using the memory registers such as M2. You can even lock a window to the indirection of its own memory register (e.g. {m2}) which might be useful to step through a linked list (in conjunction with the Esc key to update the window each time).

### Alt-W *Widen Window*

Splits a window horizontally i.e. makes it wider or narrower depending on its current state; this may hide or uncover another window. You would normally use this to set up the display as you like it and then save the set-up with the Save Preferences command. It can be useful at any time, though, if you would like to see more information in a window or you need another window.

This command has no effect on window 1.

### Shift-. *Open View*

Creates a new view on the current window and numbers it accordingly. The type of this view will be the same as the previous one if possible.

The display will be numbered xa, xb, xc, xd etc. where x is the number of the window e.g. if you stack a new display on window 2, it will be numbered 2b with the original display being numbered 2a. Remember, though, that there is only one memory register per window, but you can lock each display to a different expression. This gives a tremendous amount of flexibility.

This command does not work on a zoomed window.

The associated memory register is bound to the top view only, although all locks on all views are re-calculated where necessary.

### Shift-, *Close View*

Removes the visible display from the current window's display list, unless there is only one display attached to this window, in which case the command does nothing. If you close a view on a source window, the source file will be removed from memory and a disassembly window will replace the closed source view.

All other displays attached to this window will be re-numbered if necessary i.e. if you remove display 2c from (2a, 2b, 2c, 2d), 2d will be re-numbered to be 2c.

This command does not work on a zoomed window.



### . and , *Next/Previous View*

These two commands allow you to cycle through views that have been stacked onto a window. Pressing . (full stop or period) cycles forward through the available displays whilst , (comma) cycles backwards. Both will roll round in a loop.

For example, say you have 3 displays stacked on window 4 (4a Source, 4b Memory and 4c Disassembly) and you are currently displaying 4b Memory. Press . and 4c Disassembly will appear, press . again and you will see 4a Source.

These commands do not work on a zoomed window.

### Esc

Pressing Esc will update all the window displays, if necessary and re-calculate the addresses to which any windows and views are locked.

This can be very useful in many cases; for example say you have window 3 locked to {m5} (the address pointed to by window 5) and you then scroll through window 5. Normally this will not update window 3. However, all you have to do is to press Esc when you want to update window 3 (and all the other windows).

## *Screen Switching*

Mon uses its own screen display and drivers to prevent interference with a program's own screen output. To prevent flicker caused by excessive screen switching when single-stepping the screen display is only switched to the program's after 20 milliseconds, producing a flicker-free display while in the debugger. In addition the debugger display can have a different screen resolution to your program's if using a colour monitor.

### Control-O *Other Screen Mode*

This cycles the screen mode of Mon between the available screen modes (when using a colour monitor). It has no effect when using a high resolution mono monitor.

On the Falcon this will switch to an 80 column, four colour mode which needs considerably less RAM than a higher bit-plane screen mode. When in use on a TV or 'standard' style monitor, the debugger will inquire as to whether or not you require an overscan and/or interlaced display. Please note that selecting overscan might cause some of the display to be lost off the left and right hand edges of the display.

After changing screen resolutions Mon re-initialises window font sizes and positions to the initial display. This will not affect the screen mode of the program being debugged.

As Mon has its own idea of where the screen is, what mode it is in and what palettes to use you can use Mon to actually look at the screen memory in use by your program, ideal for low-level graphics programs.

Please note that if your program changes screen position or resolution via the XBIOS or the hardware registers, it is important that you temporarily disable screen switching using Preferences while executing such code otherwise Mon will not notice the new attributes of your program's screen.

When a disk is accessed, when loading or saving, the screen display will probably switch to the program's for the duration of the operation. This happens in case a disk error occurs, such as write-protected or read errors, as it allows any GEM alert boxes to be seen and acted upon.

# *Breakpoints*

## U *Run Until*

This now additionally can take a general breakpoint specifier ( n, =, *, or ?) as with the Alt-B command.

## *Loading & Saving*

### B *Load Binary File*

This will prompt for a filename and an optional load address (separated by a comma) and will then load the file where specified. If no load address is given then memory will be allocated from the system. M8 will be set to the start address of the loaded file and M9 to the end address. Please note that this is a change from previous versions of Mon, where M0 and M1 were set to the start and end addresses of the loaded file.



### S *Save Binary File*

This will prompt for a filename, a start address and an (inclusive) end address. To re-save a file recently loaded with the Load Binary File command

```
<filename>,M8,M9
```

may be specified, assuming of course that M8 and M9 have not been re-assigned.

### A *Load ASCII File*

This powerful command allows an ASCII file, normally of source code, to be loaded and viewed within Mon. This can be loaded into window 2 or window 4. If the loaded program has line number information relevant to this source file, you will be able to use line number operators on this display to step through the source code, set breakpoints within it etc.

A new view on this window will be opened if the window already contains an ASCII file, otherwise the text will replace the current window. You can unload a source window using the Close View command.

The source window will be locked automatically to the PC. As you single-step a disassembly window the contents of the source window will be updated accordingly. The top line of the display represents the current high level source line which is being executed.

Memory for source code displays is taken from the system so sufficient free memory must be available.

### E *Executable file to use*

This command is used to just load the symbol table & line number information from an executable file, *without* loading the executable itself. The command is ideal for debugging desk accessories and for programs loaded by others as overlays. The filename specified may be optionally followed by the address of the text segment which is to be assumed.

# Miscellaneous

## Control-P *Preferences*

A new feature has been added to the preferences when being used on a Falcon. This appears in the preferences display as `Use own screen mode (Y/N)`. When selected and then saved as a preferences option, the next time the debugger is loaded it will use its own screen rather than by cloning a copy of the current display for the application which is being debugged. This can save significant amounts of screen display memory, which on machines with smaller RAM configurations may be most useful.

The auto-resident version of the debugger now looks for a preferences file called MON.INF to be located within the current directory once it is loaded. This is normally 'C:\' (or 'A:\' on floppy disk based systems). This means that the debugger can use its alternate screen on the Falcon without first appearing in a low resolution screen.

*Start at label*

When an executable file is loaded normally Mon stops at the first location in the program. If a different label is specified using this option (e.g. `REF0001` for HiSoft BASIC), then the program will instead be stopped at that point; this means you can start debugging at the start of your code, rather than the going through the compiler's startup code.

## Q *Query (read) port*

Normally Mon will not let you read the hardware ports directly (to prevent 'upsetting' the system by reading from a number of areas at once); however you can achieve this by using the Q command. You will be prompted to enter the address you wish to read. The byte value read from this address will then be displayed. To access the memory a word or long word at a time you should follow the address by `W` or `L` respectively. Please note that careless use of this command can result in a bus error or even a complete system crash.

## T *Transfer to (write) port*

Normally Mon will not let you write to the hardware ports directly, however you can achieve this by using this command. You will be prompted for the data to transfer, of the form

```
<address>,<data>[,<size>]
```



Normally the single byte value data will be written to port address although this may be over-written by using the optional size field which must be either W or L. Please note that careless use of this command can result in a bus error or even a complete system crash.

## C *Compare Memory*

This command compares two areas of memory; you will be prompted for start and end addresses of the first block, together with the start address of the second block. If the two blocks differ, windows 2 and 3 will be placed at the first difference in the first block, whilst windows 4 and 5 will be placed at the equivalent place in the second block. The N command (Find Next) may then be used to step forward through differences, based on windows 3 and 5.

# Command Summary

*Window Commands*

| | |
|---|---|
| Alt-A | Set Address |
| Alt-B | Set Breakpoint |
| Alt-E | Edit View |
| Alt-F | Font Size |
| Alt-G | Goto Source Line |
| Alt-L | Lock to Expression |
| Alt-P | Print Window |
| Alt-R | Register Set |
| Alt-S | Split Window |
| Alt-T | Change Type |
| Alt-W | Widen Window |
| Alt-Z | Zoom Window |
| Control ← | Reduce Register Window Height |
| Control → | Increase Register Window Height |
| Shift-. | Open View |
| Shift-, | Close View |
| . and , | Next/Previous View |
| Esc | Update all Windows |

*Screen Switching*

| | |
|---|---|
| V | View Other Screen |
| Control-0 | Other Screen Mode |

*Breakpoints*

| | |
|---|---|
| Control-A | Breakpoint After |
| Control-B | Simple Breakpoint |
| Control-D | BDOS Breakpoint |
| Control-K | Kill Breakpoints |
| Alt-B | Set Breakpoint |
| U | Run Until |
| Help | Show Help and Breakpoints |

*Loading and Saving*

| | |
|---|---|
| Control-L | Load Program |
| A | Load ASCII File |
| B | Load Binary File |
| E | Use New Executable |
| S | Save Binary File |

*Executing Programs*

| | |
|---|---|
| Control-R | Return to program / Run |
| Control-S | Skip Instruction |
| Control-T | Trace Instruction |
| Control-Y | Single-Step |
| Control-Z | Single-Step |
| R | Run (various) |

*Searching Memory*

| | |
|---|---|
| G | Search Memory (Get a sequence) |
| N | Find Next |

*Miscellaneous*

| | |
|---|---|
| Alt-0 or 0 | Show Other Bases |
| Control-C | Terminate Process |
| Control-E | Re-install breakpoints |
| Control-P | Preferences |
| C | Compare Memory |
| D | Change Drive & Directory |
| H | Show History Buffer |
| I | Intelligent Copy |
| L | List Labels |
| M | Modify Address |
| P | Disassemble to Printer/Disk |
| Q | Query (read) port |
| T | Transfer to (write) port |
| W | Fill Memory With |
| Shift-Alt-Help | Interrupt Program |



# *WERCS Improvements*

There have been some minor additions to the HiSoft resource file editor. Versions of WERCS from 1.25 onwards now support the Atari 3D button system that has become a feature of all versions of the operating system which contain the AES from version 3.4 onwards.

## *Using the 3D effects*

These additions take the form of the top three items which now appear in the WERCS Flags menu. These new status flags are called *indicator*, *activator* and *background*. To fully appreciate the way in which these new features look and operate, it will be probably be necessary to see them and use them, but in principal the older style of Atari resource button can now be made to look like a computer style key which appears on the screen as a light grey shadowed button against a grey background.

When a button is pressed (using the mouse), what happens on the screen to its appearance will depend on if it is defined to be an activator or an indicator. The background option provides nothing more than an area behind the 3D buttons filled using the same light grey colour.

To make a 3D option appear on the screen it is simply not enough to select it from within WERCS, it is also necessary to select any standard, non-white, fill colour! Failure to do this will result in the button (or background) appearing on the screen as a standard white filled GEM area of the standard variety. Any attempt to use a 3D option on a machine which does not support the 3D effects will also result in a standard GEM style button appearing on the screen.

## *Indicator*

An indicator is a 3D style button which will appear on the screen as a light grey button with black text inside it. When an activator is selected, the button will provide the illusion of being physically pressed, as if by your own finger, by moving downwards and slightly to the right of its original position, it will also change colour by turning to a dark grey with white text inside it. This button should be used for status displays and for traditional radio buttons.

## Activator

At first sight an activator will look identical in appearance to an un-selected indicator button. The only difference is that when depressed it will appear to move but it will not change colour. This style of button would normally appear in place of a standard GEM selector button, but should not be used in a situation where its physical appearance indicates a status of some form. This is because an activated button does not become illuminated or darkened in any way and its change in physical position alone will not provide adequate visual feedback to a careless or unobservant user.

## Background

The background option does not really perform any true function as such, it merely provides a compatible background against which informative text and/or the 3D buttons can be displayed. In software which chooses to use the 3D effects, all dialog, control or alert boxes should use a 3D background to keep a consistent theme throughout the application.

The only other point to note is that while the external physical dimensions of the background are the same as for any other dialog or alert box, it should be noted that a small portion of the internal area is lost around all four of the edges due to the fact that a bevelled edge is added to the display. This form of relief helps to add to the illusion of the box being displayed in the foreground against the application which is put temporarily into the background.



# Additional AES and VDI Library Calls

Although the following calls have been part of the Atari's operating system for some years they have only been added to the HiSoft BASIC libraries with the 2.10 release.

## Additional GEMAES routines

### FUNCTION appl_trecord(BYVAL eventrec&, BYVAL num)

This function records a series of user actions which may then be 'played back' using the `appl_tplay` function. `eventrec&` is the address of an area of memory where the details of the events will be stored. 4 words are required for each event recorded and num gives the number of events to store. The data for each event consists of 1 word of zero, 1 word indicating the type of event and two words giving event-specific information. The different events are as follows

| Type | Description | Event-specific information |
|---|---|---|
| 0 | timer event | The two words (really a long word) together give the elapsed time in 200ths of a second; so the first word is how many 65536 200ths of a second have elapsed while the final word is the remainder of the elapsed time in 200ths of a second. |
| 1 | button event | The next word is the button state (1 for down) and the final word is the number of clicks. |
| 2 | mouse event | The next word is the x co-ordinate and the final word the y co-ordinate. |
| 3 | keyboard event | The next word is the key code and the final word is the key shift state. |

### *SUB appl_tplay(BYVAL eventrec&, BYVAL num)*

This function plays back a series of user actions that have been recorded using the `appl_trecord` function at address `eventrec&`. num events are played back.

## *Additional GEMVDI routine*

### *SUB vq_curaddress( VARPTR x%, VARPTR y%)*

This sub-program returns the current TOS cursor position in `x%` and `y%`. The top left corner is (1,1).